Win32 Multithreaded Programming

By Aaron Cohen & Mike Woodring
1st Edition December 1997
1-56592-296-4, Order Number: 2964
724 pages, $49.95, Includes CD-ROM

Chapter 1
Win32 Multithreaded Programming

In this chapter:

- What Is Multithreaded Programming?
- Why Write a Multithreaded Program (Why Use Threads)?
- When Not to Use Threads
- Making the Transition to Multithreaded Programming

If there be no great love in the beginning, yet heaven may decrease it upon better acquaintance, when we are married and have more occasion to know one another: I hope, upon familiarity will grow more contempt.

---William Shakespeare

The Merry Wives of Windsor

While multithreading has long been available on mainframes and workstations, it is a new capability for personal computers. Prior to the first release of Windows NT, 16-bit versions of the Microsoft Windows operating system provided only a crude form of multitasking known as cooperative multitasking. With cooperative multitasking, all programs needed to be "good citizens" and share the CPU with other programs to enable the user to run more than one program at the same time. Unfortunately, most software was not always so well behaved, and the result was that running more than one program at a time was often more trouble than it was worth. All this changed for the better with the 32-bit Windows operating systems, Windows NT and Windows 95, which support preemptive multitasking and multithreading. Applications can now be written pretty much as if they are the only program running on the system, and the operating system ensures that all of the programs share the CPU and behave themselves.

With the new multithreading capabilities came new challenges for programmers. Most PC programmers were raised on DOS and other simple operating systems. Making the transition to Win32 programming and taking full advantage of the advanced features of the

32-bit operating systems can be difficult. Programmers who have not had experience with more advanced systems may not have been exposed to even the basic concepts of multitasking and multithreading.

In order to provide good grounding for the reader, this chapter will explain what multithreading is, and why you would want to use it. Because multithreading is not a solution to every problem, we will then discuss situations in which you would not want to use multithreading. The chapter ends with an introduction to the basic mindset you must have in order to write correct multithreaded programs.

What Is Multithreaded Programming?

So what is multithreaded programming? Basically, multithreaded programming is implementing software so that two or more activities can be performed in parallel within the same application. This is accomplished by having each activity performed by its own thread. A thread is a path of execution through the software that has its own call stack and CPU state. Threads run within the context of a process, which defines an address space within which code and data exist, and threads execute. This is what most people think of when they refer to "multithreaded programming," but there really is a lot more to programming in a multithreaded environment.

Good multithreaded programming involves more than simply creating additional threads. We can loosely divide the issues into two categories. The first issue involves writing your software to use multiple threads in a useful and efficient manner. Carefully written multithreaded programs should be superior to single threaded designs in terms of execution time, user responsiveness, architecture, or all three.

The second issue is awareness of the operating system's rules that govern the behavior of your program while it's running, as well as understanding how your program interacts- directly or indirectly- with other programs running at the same time. You need to understand not just how the operating system will treat your program, but how it will treat your program when other programs are running at the same time. Likewise, understanding the impact your program has on other programs trying to run at the same time is just as important.

Becoming knowledgeable and adept at both aspects of multithreaded programming will be crucial to your success as a programmer on Windows 95 and Windows NT. This book will cover both aspects of writing good multithreaded programs.

Why Write a Multithreaded Program (Why Use Threads?)

So why would you want to add extra threads to your programs? After all; you've been getting by just fine without support from Windows for incorporating multiple threads into your programs. Why start now? It turns out that there are some distinct advantages to using multiple threads in your programs. Some of those advantages include:

- Increased parallelization. Very often, programs need to accomplish more than one task as a result of some initial event (like a user pressing a button, or a service request coming in from another machine in the network). If these tasks are essentially independent activities, the performance of an application can be improved by having a separate thread take care of performing each activity. In a single threaded application, the total length of time required to accomplish three tasks is the sum of the times that it takes to accomplish each task serially, as shown in Figure 1-1.

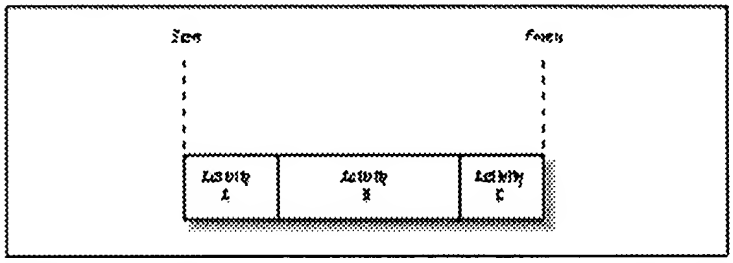

Figure 1-1: Time required for three tasks in a single threaded application

- Faster processing. In a multithreaded program, the total length of time required to accomplish all three tasks is just the time it takes to complete the longest of the individual tasks. This is illustrated in Figure 1-2.

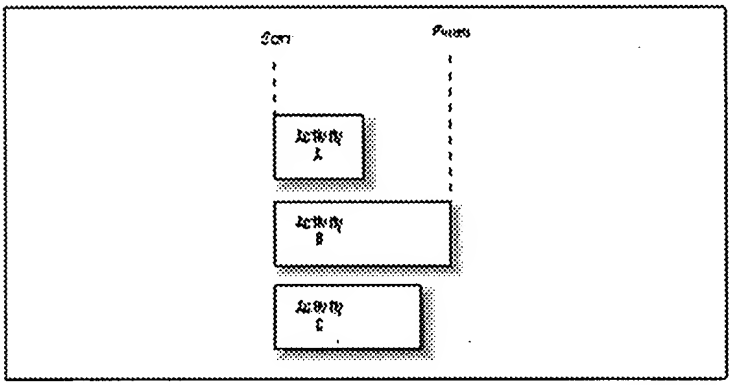

Figure 1-2: Time required for three tasks in a multithreaded application

- Maximum parallelization. For most multithreaded applications, where each thread spends a large faction of its time waiting for an I/O operation to complete, or a kernel object to become signaled, maximum parallelization of the kind illustrated by this diagram can be achieved. In other words, if the thread performing Activity A spends a significant portion of the time waiting for some I/O operation to complete, the thread performing Activity B can accomplish useful work while thread A is blocked. And on multiprocessor machines, the operating system will be able to execute threads truly concurrently to one another by allowing one thread to run on each CPU.

- Simplified design. A well-designed multithreaded program can use threads to actually simplify the design of the program by dedicating a unique thread to each well-defined, independent job. For example, an audio playback program can be designed to use one thread for reading compressed audio from a disk file, a separate thread for decompressing the audio stream, and a third thread for playing back the audio data to the speakers. This design allows the file-reading thread, which will spend a large portion of its time blocked waiting for disk I/O operations to complete, to read ahead

from the input file while the second thread is decompressing audio data that was previously read from disk. Likewise, the playback thread can be playing the uncompressed audio stream to the speaker smoothly without concern for the other two activities. A single threaded application would have to interleave small portions of disk reads, decompression, and playback in an unnecessarily complicated manner in order to present equally smooth- sounding audio to the user.

- Increased robustness. By using multiple threads within a program, it's possible to increase the robustness of an application by isolating critical subsystems into their own thread (or threads) of control. For example, you might want to ensure that even if subsystem A fails as a result of invalid user input, subsystem B (the thermonuclear device temperature monitoring thread) can continue to run unabated.

- Increased responsiveness to the user. By using multiple threads to separate the user interface portions of your program from the rest of your program, you can increase the responsiveness to the user, even if the program is "busy" doing something. For example, a single threaded Internet browser application that wants to allow the user to cancel bringing in data from a large web page would have to periodically call PeekMessage or devise some other method of interrupting the data transfer. By performing network data transfers on a background thread, the user interface thread running at a higher priority can instantly react to the user's desire to cancel the lengthy operation.

- Better use of the CPU. A lot of the work done by Windows programs happens in short bursts in between long waits for something to occur. Waiting for a block of data to be read from the CD-ROM drive, or for a buffer of data to be written to a COM port, are both examples of activities that include plenty of built-in waits for devices operating at speeds much slower than the CPU to finish a particular job. By performing these activities on individual threads, the operating system can do a better job of keeping the CPU busy doing useful work while I/O bound threads are waiting for these slow devices to finish doing something useful.

There are other good reasons to incorporate the use of multiple threads in your applications, but these are the most fundamental.

When Not to Use Threads

Just because an operating system supports the use of multiple threads in a program doesn't necessarily mean you should have multiple threads in your program. In fact, at times there are disadvantages to using multiple threads in order to accomplish a job. When most programmers discover multithreaded programming for the first time, they're almost giddy with excitement. Every problem solved by their programs is broken down into its own thread. And who can blame them? It's just plain fun to watch your program work on more than one chore at a time, like so many little automated robots running around a factory floor busily doing their assigned job without regard for other activities going on in the plant. While there are many advantages to using multiple threads in your programs, adding additional threads also introduces complexity and the possibility of encountering new classes of errors (deadlock, starvation, etc.) that are not part of the landscape of single threaded programming. Here are some guidelines to help you decide when not to incorporate the use of multiple threads in your application:

- You don't have a really good reason. This should be the first yardstick against which you measure the need to incorporate a thread in any program. Very often, the novice multithreaded programmer will not have a good reason to be adding a new thread into a program-it just happens to be fun. And what's the harm? The answer is "potentially, lots." The inclusion of just one extra thread in an otherwise single threaded application brings with it a whole nest of design, implementation, and debugging problems that need to be considered.

  For example, you shouldn't divide a job between two threads when each thread could not otherwise stand on its own. In other words, if two threads are involved in doing job X, and the absence of one thread or the other would preclude job X from being done, you probably don't have a good reason to involve two separate threads.

- The operating system overhead involved with scheduling and otherwise dealing with a thread, when taken together with the frequency with which the thread does its job, outweighs the amount of work actually performed by that thread. In other words, if a thread does only a little bit of work, but does it very often, the overhead incurred by the operating system as it tries to schedule your thread to run many times per second can outweigh the other benefits you might have sought to gain by introducing the thread into your program. Keep in mind that this does not mean that each thread has to do lots of work when it runs. Having a thread that runs very infrequently, and that does a little bit of work when it does run, is not going to cause system performance to drop. It's threads that wake up and run very often, and that do only a little bit of work when they do run, that will degrade the performance of not only your application, but the machine as a whole.

For example, the overhead of additional threads will outweigh the actual work performed when threads are used in an essentially serial manner. If the threads in your application always run synchronously with respect to one another, the overhead involved with scheduling each thread to run and communicating information between each thread is not justified.

There are some other more subtle reasons why including multiple threads in the design of a program might not be the best for the success of your application. Because a multithreaded program involves a new class of problems, the skill sets of the developers actually implementing the software play into the decision to incorporate multiple threads in an application. Just because the Win32 API documents all the functions needed to write multithreaded software doesn't mean that a programming team has everything they need to successfully develop a good multithreaded application.

Making the Transition to Multithreaded Programming

As you make the transition from a single threaded programming environment to a multithreaded one, you'll need to increase the scope with which you look at a problem. It will no longer be enough to know how to manipulate data structures or to correctly pass parameters in order to accomplish some useful task. For example, when you're looking at a function's implementation, you need to understand not just whether or not the function does its job right, but whether or not the function does its job right in the presence of multiple

threads.

Your major concern is whether or not multiple threads might execute the function at the same time. If the answer is yes, you need to make sure the function does its job right no matter when each thread executes the function. Very often, answering this question involves more than it would first appear. If the function modifies data that is visible to more than one thread, you need to understand not just whether or not this function takes the appropriate precautionary measures before accessing the data, but whether or not all other threads accessing the data also take the appropriate steps required to properly synchronize access to the data.

To illustrate just one of the issues facing you as you move into the world of multithreaded programming, assume that you have a Windows 3.x application that maintains a singly linked list of available GADGET objects that have been pre-allocated, and that the pointer to the first free GADGET is kept in a global variable:

```
typedef struct tagGADGET {
{
   // Fields of GADGET structure here.

   // Pointer to next GADGET in the list.
   struct tagGADGET *pNext;
}
GADGET, *PGADGET;

PGADGET gpFreeGadgets = NULL;      // Global pointer to list of free GADGET
                                   // structures.
   // The list is empty if gpFreeGadgets == NULL.
```

Figure 1-3 illustrates the linked list and the pointer to its first free GADGET structure. You would probably have one function for removing a node from the list of available GADGET structures, and another function for placing a GADGET structure that's no longer needed back on the free list.

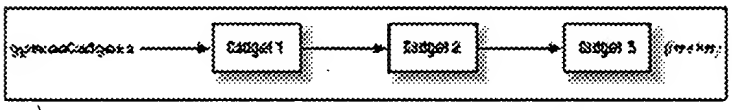

Figure 1-3: Initial state of the gadget free list

```
PGADGET GetAGadget( void )
{
   // Grab the GADGET structure at the head of the free list (which might
   // be empty).
   //
   PGADGET pFreeGadget = gpFreeGadgetss;

   if( pFreeGadget ) {
      // The list wasn't empty, so remove this node from the free
      // list by setting the list head (gpFreeGadgets) to point to the
      // next node in the list.
      //
      gpFreeGadgets = gpFreeGadgets->pNext;
   }

   return(pFreeGadget);
```

```
}

void FreeAGadget( PGADGET pGadget )
{
   // Put the given GADGET structure back on the free list by placing
   // it at the head of the list.
   //
   pGadget->pNext = gpFreeGadgets;
   gpFreeGadgets = pGadget;
}
```

If you were to port this application to Windows 95 or Windows NT and separate the work into two threads within your program-one that pulled GADGETs off the free list and another that put GADGETs back on the free list-you're bound to run into trouble. The problem is that the thread taking GADGETs off the list and the thread putting GADGETs back on the list are scheduled independently by the operating system. Consequently, each thread might be part of the way through one of the above functions when its execution is interrupted and the other thread is allowed to run. Based on timing, the above code will start to exhibit a bug that, because it is intermittent, can be very difficult to diagnose and correct.

For example, assume that thread B (the thread freeing GADGETs) calls FreeAGadget and makes it past the first assignment statement:

```
pGadget->pNext = gpFreeGadgets;
```

just before the operating system takes control away and lets thread A (the thread pulling free GADGETs off the free list using GetAGadget) run. At this point, the GADGET structure being placed back on the free list by thread B has a pNext member that points to the GADGET structure referenced by the gpFreeGadgets structure, as Figure 1-4 illustrates. If thread A were to call GetAGadget at this time, a pointer to the same GADGET structure pointed to by gpFreeGadgets would be returned, and thread A would merrily go about using that GADGET structure, as shown in Figure 1-5. When thread B continues to execute, it will pick up where it left off, just after reading the value of gpFreeGadgets, which has since been updated to point to some other GADGET structure. Unfortunately, thread B doesn't know this, so it continues by updating gpFreeGadgets to point to the GADGET structure being freed. The problem is that this GADGET structure's pNext member has already been set to point to the GADGET structure that was previously at the head of the free list before thread A called GetAGadget. At this point, as you can see by examining Figure 1-6, the list is in a state it should never be in: a GADGET structure that's on the free list (what is now the second node in the list) is also being used by another thread in the program that believes it has properly allocated that GADGET structure for its own use.

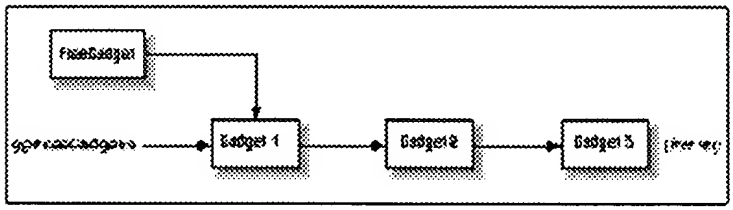

Figure 1-4: Thread B begins to put a gadget back on the free list

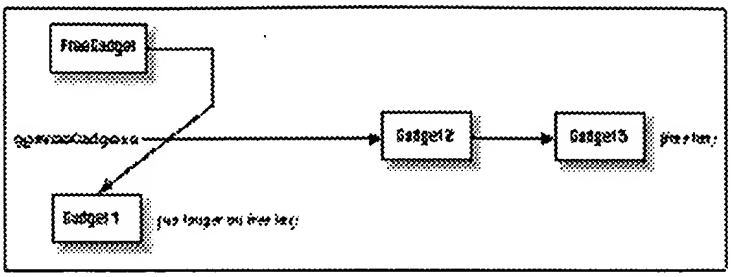

Figure 1-5: Thread A takes a gadget off the free list

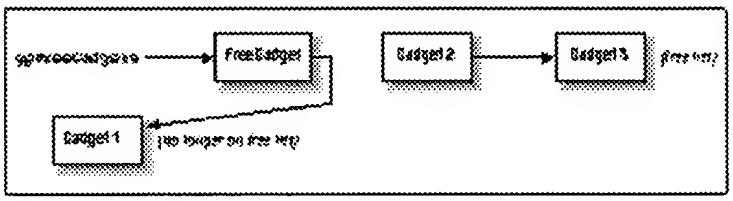

Figure 1-6: Thread B finishes putting a gadget back on the free list, placing the list in an inconsistent state

As you can see, what used to be a relatively simple piece of code inherits new complications as a result of moving from a single threaded application to a multithreaded one. In addition to making sure the algorithms embodied in your application are correct, you'll have to worry about the environmental effects of the algorithm running within a multithreaded program. In addition to simply making sure that you can manipulate data structures such as the ones shown above, you must now consider and design for issues such as starvation, deadlock, and proper synchronization. It will be up to you as a programmer to protect yourself against these sources of error. Your design, implementation, and debugging skills will each need to be updated in order to thrive in a multithreaded programming environment. This book will give you the tools necessary to improve those skills and to incorporate the use of multiple threads into your application safely and effectively.

Back to: Win32 Multithreaded Programming

O'Reilly Home | O'Reilly Bookstores | How to Order | O'Reilly Contacts
International | About O'Reilly | Affiliated Companies